FICHA 1
CONSOLIDAÇÃO DE CONCEITOS DE PROGRAMAÇÃO EM C

1.1 Objetivos

Esta ficha tem por objetivo geral fazer uma revisão de conceitos importantes da linguagem C, que foram ensinados na disciplina de Programação de Computadores:

- Rever os principais conceitos relacionados com variáveis, controlo de fluxo, estruturas de dados e funções;
- Rever os principais conceitos relacionados com a manipulação de estruturas de dados básicas – tabelas, estruturas e ficheiros;
- Tornar o aluno consciente da importância da indentação na escrita de um programa, ao torná-lo mais legível (por outros e pelo próprio), reduzindo a possibilidade de ocorrência de erros ("*bugs*");
- Apresentar outras regras importantes a ter em conta na escrita de programas, como por exemplo a utilização de comentários para aumentar a legibilidade do código, reduzir o número de "*bugs*" introduzidos na escrita de código e reduzir os tempos de desenvolvimento;
- Relembrar alguns conceitos básicos sobre variáveis e ponteiros para variáveis;
- Rever os métodos de passagem de parâmetros a funções, nomeadamente passagem por valor e por ponteiro.

1.2 Instruções de controlo de fluxo: ciclos

Um aluno com conhecimentos elementares ao nível da programação deverá ser capaz de utilizar as instruções básicas para a implementação de ciclos, **`for / while / do`**, para executar tarefas sequenciais, como percorrer *strings* e tabelas. Esta é de facto a utilização mais comum para estas instruções.

Considere o seguinte exemplo:

```
char nomes[20][60];
. . .
int i;
for (i=0; i<20; i++) {  /*  imprimir uma tabela com 20 nomes  */
      printf("O nome do estudante %d é %s\n", i+1, nomes[i]);
}
```

O ciclo **`for`** pode ter uma utilização anormal dos seus parâmetros de iniciação, teste ou incremento. Esta é uma particularidade usada por alguns programadores para escrever código compacto. Sugere-se no entanto que o aluno não recorra a este tipo de programação "compacta", uma vez que esta aumenta a probabilidade de ocorrência de erros durante a programação.

Qualquer um dos ciclos **`for`**/**`do`**/**`while`** pode ser convertido num dos outros. Considere os seguintes exemplos:

Ciclo **`for`** convertido em **`while`**

```
for (A; B; C) {
  D
}
```

```
A;
while (B) {
  D
  C;
}
```

O subprograma da esquerda e o da direita fazem exatamente o mesmo independentemente do que seja A, B, C ou D (instruções ou subprogramas). No entanto, a clareza do código é obviamente diferente. Note que o subprograma da esquerda não é obrigatoriamente mais simples que o da direita. Tudo depende da complexidade das partes A, B, C e D.

Ciclo **`while`** convertido em **`for`**

```
while (E) {
  F
}
```

```
for (;E;) {
  F
}
```

Neste caso, o subprograma da esquerda faz mais sentido que o da direita, uma vez que é mais legível.

Muitas vezes é necessário optar por utilizar o ciclo **`do`**/**`while`** em vez do ciclo **`while`** "normal". O ciclo **`do`**/**`while`** é usado quando um ciclo deve ser executado pelo menos uma vez, independentemente de qualquer condição. Repare que a condição que avalia a continuidade do ciclo só é verificada no fim. É possível no entanto utilizar um ciclo **`while`** para se conseguir obter o mesmo resultado que o ciclo **`do`**/**`while`**:

```
do {
  G
} while (H);
```

```
while (1) {
  G
  if (!H) break;
}
```

Os dois subprogramas fazem na prática o mesmo, mas o da esquerda é mais legível e traduz uma programação mais estruturada.

Uma das utilizações mais comuns para os ciclos traduz-se na utilização de vários ciclos embutidos (uns ciclos dentro de outros). Este embutimento tende a criar alguma confusão, sobretudo a um aluno que não domine perfeitamente o comportamento das instruções **`for`**/**`do`**/**`while`**. O raciocínio aplicado à construção de um encadeamento de 2 ciclos **`for`** assemelha-se muito ao usado na especificação de um somatório de somatórios. Por exemplo, se quisermos calcular a soma de todos os elementos de uma matriz bidimensional (`m`×`n`),

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_{ij} \ ,$$

precisamos de encadear dois somatórios:

```
#include <stdio.h>

#define LINHAS  4     /* Linhas */
#define COLUNAS 3     /* Colunas */

int main() {
  int a[LINHAS][COLUNAS] = {1,2,1, 2,1,2, 1,2,1, 2,1,2};
  /* vai somar todos os elementos da matriz */
  int s = 0;
  for (int i = 0; i < LINHAS; i++)
       for(int j = 0; j < COLUNAS; j++)
          s += a[i][j] ;
  printf("A soma vale %d\n", s);
}
```

Quando calcula a soma de duas matrizes bidimensionais `m×n` precisa de encadear dois ciclos for:

$$s_{ij} = a_{ij} + b_{ij} \ \ (i=0..m\text{-}1,\ j=0..n\text{-}1).$$

O cálculo do produto de uma matriz bidimensional *a*(`mxn`) por uma matriz *b*(`n×p`) é uma matriz *s*(`m×p`):

$$s_{ik} = \sum_{j=0}^{n-1} a_{ij} b_{jk} \ , (i=0..m\text{-}1,\ \ k=0..p\text{-}1)$$

e requer três ciclos **`for`** encadeados.

Sugestões: procure escrever um programa em C que calcule a soma de duas matrizes. Tendo cumprido essa tarefa, escreva um programa que calcule o produto de duas matrizes bidimensionais.

1.3 Estruturas

As **Estruturas** (*structs*) são uma forma conveniente de se agrupar um conjunto de dados associados a um determinado objeto ou entidade. Existem vários exemplos da vida real onde são utilizados "registos" ou "fichas" que representam uma determinada estrutura: por exemplo o BI com vários campos a identificar uma pessoa e os seus dados pessoais; o livrete do automóvel, contendo vários dados relativos ao automóvel a que corresponde; a carta de condução; o cartão de contribuinte; o cartão Multibanco; a ficha de aluno na secretaria; etc.

Exemplo de definição de uma estrutura:

```
struct Arco {
  int no_origem, no_destino;
  float custo;
};
```

Exemplo de declaração de uma variável de tipo `struct Arco` (note que aqui poder-se-ia ter utilizado um typedef para a palavra reservada `struct` poder não ser usada na declaração):

```
struct Arco a;
```

Manipular uma estrutura é ligeiramente diferente da manipulação de uma variável simples (ex. um inteiro). Para além de identificar qual é a estrutura que queremos aceder, teremos também de especificar qual é o campo dentro da estrutura que pretendemos manipular.

Exemplo de uma atribuição simples:

```
i=5;
```

Segue-se um exemplo de declaração de uma estrutura e de atribuição de valor aos seus membros.

```
struct TData{
  int Dia,Mes,Ano;
};

struct TData dataNascimento;

dataNascimento.Dia = 19;
dataNascimento.Mes = 10;
dataNascimento.Ano = 1970;
```

Se pretendermos copiar o conteúdo integral de uma estrutura para outra, poderemos facilmente fazer uma atribuição ao nível da estrutura:

```
TData dataNascimentoA, dataNascimentoB;

dataNascimentoA.Dia = 19; dataNascimentoA.Mes = 10; dataNascimentoA.Ano = 1970;

dataNascimentoB = dataNascimentoA;
```

1.4 Tabelas

As tabelas (também designadas por *arrays* ou vetores) são uma forma adequada de se armazenar um conjunto (possivelmente grande) de dados do mesmo tipo. O acesso a cada um dos dados é feito através da indexação da tabela. Exemplo: **`a[i]`** representa o *(i+1)*ésimo valor armazenado na tabela **`a[]`**.

Um dos problemas típicos que ocorrem frequentemente na manipulação de tabelas resulta da tentativa de se aceder a um elemento para lá do último elemento da tabela. O compilador C não verifica a validade do índice da tabela. Este tipo de erros é tanto mais comum quanto mais complexa for a indexação do elemento da tabela, por exemplo:

```
  a[i+(j-1)*6*(y+7)] = 3;
```

Este tipo de indexação pode ter consequências nefastas, dependendo dos valores que podem tomar as variáveis **`i`**, **`j`**, **`y`** e dependendo também da dimensão da tabela. É absolutamente necessário que o programador valide os limites de indexação da tabela para que uma atribuição como a do exemplo acima não se torne num *bug* muito difícil para o programador resolver. É preciso verificar o valor mínimo que o índice da tabela pode tomar – nunca pode ser inferior a zero – e o valor máximo – nunca poder ser superior ao número de elementos da tabela menos 1.

Podemos utilizar um bloco unidimensional de 12 inteiros para armazenar uma matriz bidimensional de 3×4 inteiros em memória, matriz essa gerida pelo programador. Definição:

```
  int matriz[3*4];         /* definição de bloco com 12 inteiros */
  ...
  for (int linha = 0, i = 0; linha < 3; linha ++)
    for (int coluna = 0; coluna < 4; coluna ++, i++)
       matriz[linha*4 + coluna] = i;     /* equivalente a: matriz[i] = i */
```

Deverá ter um especial cuidado na verificação dos índices de forma a evitar o aparecimento de erros. O programador terá que ter a abstração suficiente para imaginar a disposição dos dados na tabela e evitar problemas. De notar, contudo, que **seria mais claro** definir, em vez de uma tabela unidimensional, uma tabela bidimensional:

```
int matriz[3][4];                    /* definição de tabela 3x4 */
```

e utilizar dois índices em vez de percorrer as tabelas como foi exemplificado.

Falta fazer uma chamada de atenção para as consequências de se cometer um engano. Vejamos um exemplo:

```
int a, b[10],c; /* definição de 2 variáveis inteiras e uma tabela com 10 inteiros */
char d;
```

O que acontece se fizermos a atribuição **`b[-1] = 0`**? (nitidamente um erro)

Dependendo um pouco do compilador, o que o programa poderá fazer é escrever em cima do inteiro `a` (porque `a` está armazenada na memória exatamente antes). Ou seja, cometemos um erro a escrever em **`b[]`** e escrevemos na variável declarada antes. Outro tipo de erro também comum: o que acontece se fizermos **`b[10]=3`** ? (**`b[10]`** é uma posição acima do último elemento da tabela **b**). Muito possivelmente escreve-se em cima da variável **`c`** (depende do compilador). "*BUG*"!

E o que acontece se fizermos **`b[14]=7`**?

O erro aqui poderá ser mais grave, uma vez que esta indexação poderá aceder a uma zona de memória para além do espaço reservado às variáveis do programa. Poderá inclusive essa parte da memória estar reservada para as instruções do próprio programa. Consequência: altera-se o programa a partir desse momento, podendo provocar uma falha tal que só é ultrapassada terminando o programa e voltando a executá-lo; e isto de cada vez que se executa um excerto de código aparentemente "inocente".

1.5 Manipulação de Ficheiros (*files*)

A gravação e leitura de dados em ficheiro são uma tarefa comum em quase todos os programas. Os dados armazenados no disco de um computador sob a forma de ficheiros podem representar imagens, texto, filmes, base de dados, músicas, apresentações, jogos, executáveis, etc.

Na maioria das vezes, pretende-se que os programas processem informação e a armazenem de uma forma mais duradoura do que o tempo de vida do programa, i.e. de forma persistente. Inevitavelmente, a maioria dos programas manipula ficheiros. Que tarefas poderá realizar com os ficheiros? Guardar informação (escrever/gravar), procurar informação (ler/reproduzir), processar informação (ler, processar, gravar), etc.

Para além do procedimento formal de manipular um ficheiro – abrir, ler/gravar, fechar –, o programador terá que ter alguns cuidados adicionais. Ao contrário das outras estruturas de dados, os ficheiros podem confrontar o programador com diversas situações de exceção. Podemos tentar abrir um ficheiro que não existe, tentar gravar num ficheiro onde não há permissões de escrita, tentar aceder a um ficheiro que deixou de existir a partir de certa altura (e.g. o utilizador retira o disco externo, ou a *pen drive*, ou o CD/DVD da *drive* durante uma leitura ou escrita), ou ainda tentar ler depois do final do ficheiro. Há que lidar com estas situações SEMPRE! Uma boa maneira de proceder:

```
#include <stdio.h>
...
char nome_ficheiro[81];  /* o nome do ficheiro só poderá ter 80 caracteres, 0..79 */
FILE *meu_ficheiro;      /* variável associada ao ficheiro a abrir para leitura */
...
printf("Introduza nome do ficheiro: ";   /* pede nome ao utilizador */
scanf("%s", nome_ficheiro);   /* le o nome */
meu_ficheiro = fopen(nome_ficheiro, "r");   /* abre o ficheiro para leitura */
if (meu_ficheiro == NULL) {   /* Algum erro na abertura do ficheiro ? */
  printf("Houve um problema a abrir o ficheiro com o nome %s\n", nome_ficheiro);
} else {   /* o ficheiro foi aberto com sucesso */
  ...      /* colocado aqui todo o código referente ao uso do ficheiro */
  fclose(meu_ficheiro);/* fecha o ficheiro quando o programa termina a sua leitura */
}
```

Neste exemplo, teve-se o cuidado de verificar se o ficheiro existe ou não (na pasta onde o programa está a correr ou na pasta indicada pelo caminho fornecido com o nome do ficheiro). Cuidados como este têm de ser uma constante no uso de ficheiros.

Há ainda a referir o acesso aleatório a ficheiros. O que é isso de acesso aleatório? Quer dizer que se pode "saltar" diretamente para uma posição específica do ficheiro. É comum evitar ler-se a totalidade de um ficheiro quando se pretende apenas ler um único elemento situado algures a meio ou no fim do ficheiro.

Imaginemos que queremos ler o elemento 1 000 000 de uma tabela de inteiros. Como faríamos?

```
int resultado, tabela[2000000];
...
resultado = tabela[999999];  /* resultado passa a ter o valor guardado na tabela */
```

E se quisermos fazer o mesmo a um ficheiro, ler o elemento 1 000 000 de um ficheiro de inteiros?

```
int resultado;
FILE *meu_ficheiro;  /* ponteiro para o ficheiro */
...
meu_ficheiro = fopen("meus_inteiros.dat", "rb");
    /* ficheiro aberto para leitura de bytes (e não de texto) */
...
fseek(meu_ficheiro, 999999 * sizeof(int), 0);  /* posiciono-me na posição
                            exata do ficheiro onde está o elemento 1000000 */
fread(&resultado, sizeof(resultado), 1, meu_ficheiro);  /* lê-se os 4 bytes de
                um inteiro do ficheiro para os 4 bytes do inteiro resultado */
...
fclose(meu_ficheiro);   /* Não esquecer de "fechar" o ficheiro */
```

Foram omitidas neste exemplo as verificações necessárias para garantir que o acesso ao ficheiro é correto. Como se constata, é um pouco mais complicado manipular um ficheiro de inteiros do que uma tabela de inteiros. Contudo, dominando bem a manipulação de ficheiros, as duas tarefas são bastante análogas do ponto de vista concetual. Imagine que um ficheiro é uma tabela um pouco mais "trabalhosa" de manipular, nada mais.

1.6 Indentação (*indentation*): um salva vidas

Uma das possíveis fontes de *bugs* criados involuntariamente por programadores pouco experientes é a falta de organização visual do código. Quando o programador está na fase de tentar aprender a sintaxe de instruções, a tentar criar os seus primeiros algoritmos, ou simplesmente a tentar fazer um programa que cumpra determinados objetivos, raramente dá atenção a textos, como este, que o tentam alertar para o facto de ser **vital indentar o código**. Aliás, **é vital indentar, comentar e documentar todo o código que se faz**.

A indentação consiste em escrever o código fonte de forma a mostrar visualmente a hierarquia e a estrutura do código. A indentação deve-se aplicar na definição de variáveis, na estruturação de uma função ou na estruturação de um ciclo ou em qualquer bloco de controlo de fluxo (e.g., **if**, **if-else**, **switch**, etc.). Considere os dois exemplos a seguir relativos à definição de variáveis:

```
int a,b,c,d,e,f;
float f1,f2,a3,f14,e15,c13,
v20,c40;
char xy,de,ef;
int g,h;
char df;
```

```
int
  a, b, c, d, e, f,
  g, h;
float
  f1, f2, a3, f14, e15,
  c13, v20, c40;
char
  xy, de, ef, df;
```

Como pode verificar, as definições à esquerda e à direita são exatamente as mesmas. Qual das duas é mais clara para si? Que lição tira deste exemplo?

Exemplo, definição de *struct*:

```
typedef struct {
int num, estado_civil;
char nome[20], data[8];
}tBI;
```

```
typedef struct {
  int  num;
  char nome[20];
  char data[8];
  int  estado_civil;
} tBI;
```

Qual das 2 definições é mais clara? Que aprendeu com este novo exemplo?

Exemplo: ciclos **for**:

```
for (a=0; a<20; a++) {
for (b=10; b<30; b+=2)
for (z=5; z>-3; z--)
d=3+z+b*a;
s+=d;
for (x=3.5; x<7.0; x+=1.5){
s+=x; }
}
```

```
for (a=0; a<20; a++) {
  for (b=10; b<30; b+=2) {
    for (z=5; z>-3; z--) {
      d=3+z+b*a;
    }
  }
  s+=d;
  for (x=3.5; x<7.0; x+=1.5) {
    s+=x;
  }
}
```

Você acredita que o código da direita faz exatamente o mesmo que o código da esquerda? Qual dos dois textos lhe permite visualizar melhor a forma como os ciclos estão embutidos? (Note que foram acrescentadas chavetas no código à direita para melhorar a clareza).

Repare que todos os exemplos de códigos indentados têm tipicamente melhor "aspeto" no que toca à estruturação. Mais ainda, o uso de linhas de código adicionais, bem como o uso de mais chavetas para evidenciar hierarquia de encadeamento melhora a legibilidade do código, reduzindo portanto a

probabilidade de o programador cometer erros na sua escrita (ex. falta de chavetas). A maneira de programar apresentada à esquerda nos exemplos anteriores é muito propícia ao aparecimento de erros de programação.

Quem me diz que não foi cometido um erro no programa da esquerda e que o que lá deveria estar seria:

```
for (a=0; a<20; a++) {
for (b=10; b<30; b+=2)
for (z=5; z>-3; z--) {
d=3+z+b*a;
s+=d; }
for (x=3.5; x<7.0; x+=1.5){
s+=x; }
}
```

```
for (a=0; a<20; a++) {
  for (b=10; b<30; b+=2) {
    for (z=5; z>-3; z--) {
      d=3+z+b*a;
      s+=d;
    }
  }
  for (x=3.5; x<7.0; x+=1.5) {
    s+=x;
  }
}
```

No código da direita nota-se uma diferença abismal para a versão anterior. Já no código da esquerda essa diferença não é tão notória.

Cada pessoa é livre de fazer a indentação ou estruturação do código que desejar. Todavia, passadas algumas décadas depois do surgimento das primeiras linguagens de programação, foram adotadas algumas regras/recomendações seguidas por milhares de programadores em todo o mundo. Por exemplo, sempre que se desce um nível na hierarquia de um programa, desloca-se o texto dois espaços (ou uma tabulação) para a direita e sempre que se volta a subir na hierarquia desloca-se o texto dois espaços (ou uma tabulação) para a esquerda (ver em cima).

Esta ficha está longe de ser capaz de ensinar todo um conjunto de regras que programadores levaram anos a aperfeiçoar para evitar *bugs* ou melhorar a clareza do código. Apenas para dar alguns exemplos, hoje em dia é assumido todo um conjunto de regras para fazer bom código:

- Há regras para dar nomes às variáveis e ao uso de letras minúsculas e maiúsculas;
- Há regras para numerar versões de código;
- Há regras sobre como validar o código;
- Há várias regras de indentação do texto do código;
- Há regras para escrever código em C portável para diferentes tipos de arquiteturas de máquinas sem envolver qualquer tipo de alteração do código (muitas restrições!);
- Há regras sobre como devem ser colocados os comentários e como os comentários mudam de linha quando são muito extensos.

Espera-se que após a leitura desta secção sobre indentação fique sensibilizado para a necessidade de se organizar adequadamente a escrita de código fonte, para não correr o risco de gastar a maior parte do seu tempo a detetar e corrigir *bugs* pouco óbvios, por vezes quase "inexplicáveis", ou compreender "aquele pedaço de código que fiz há um mês e já não entendo como foi codificado e como funciona", tão somente porque a escrita do código não foi realizada de forma clara e/ou organizada.

1.7 Introdução: relembrar conhecimentos sobre ponteiros e variáveis...

A declaração de uma variável do tipo ponteiro tem a seguinte sintaxe: `<tipo> * nomeVar;`

Uma variável do tipo ponteiro "não aponta para nada" até possuir o endereço de uma posição de memória reservada para dados do programa. Quando queremos indicar que um ponteiro não contém informação útil (i.e. que não aponta para nada), podemos fazer a seguinte atribuição:

```
int *pont;
pont = NULL;
```

Analise o código que se segue procurando relembrar os conceitos aprendidos anteriormente sobre ponteiros:

```
int a, b;
int *p_1, *p_2;

/* As variáveis a e b têm dois valores... */
a = 10; b = 20;

/* ...e existem ponteiros que "apontam para elas" */
/* (O que significará isto?) */
p_1 = &a; p_2 = &b;

/* O que irá acontecer nas linhas que se seguem? */
printf("%p, %p\n", p_1, p_2);
printf("%d, %d\n", *p_1, *p_2);
p_1 = p_2; p_2 = &a;
printf("%d, %d\n", *p_1, *p_2);
```

Os compiladores da linguagem C e C++ armazenam as tabelas linha a linha (em Fortran as matrizes são armazenadas coluna a coluna!). Assim:

```
#include <stdio.h>
#include <stdbool.h>
...
int matriz[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};     /* declaração e inicialização */
int * pont = &matriz[0][0];                          /* ponteiro para 1º elemento */

bool linha_a_linha = true;  /* Confirma (ou não) armazenamento linha a linha */

for (int linha = 0; linha < 3 && linha_a_linha; linha ++)
  for (int coluna = 0; coluna < 4 && linha_a_linha; coluna ++)
    linha_a_linha = (&matriz[linha][coluna] == &pont[linha*4+coluna]);

if (linha_a_linha) printf("\nArmazena linha a linha\n");
else printf("\nNão armazena linha a linha\n");
```

faria com que surgisse no ecrã a mensagem:

```
Armazena linha a linha
```

No entanto **não há necessidade de percorrer uma matriz bidimensional usando um ponteiro, quando é muito mais claro e menos suscetível a *bugs* usar o nome da matriz e os índices adequados**.

1.8 Passagem de parâmetros por valor e por ponteiro

Como foi referido durante o estudo das funções, a passagem de parâmetros pode ser feita por valor, ou seja, é passada uma cópia do parâmetro. Isto implica que as alterações feitas ao parâmetro dentro da função não se reflitam no parâmetro real (exterior à função chamada) uma vez que foram feitas sobre uma cópia dele.

Na prática, é bastante comum termos a necessidade de devolver múltiplos valores. Uma forma de o conseguir será alterar os valores dos parâmetros da função. Recordemos como se pode conseguir isso em C.

Imagine que pretende uma função que *deveria permitir* trocar o conteúdo de duas variáveis inteiras:

```
void Troca(int Num1, int Num2) {   /* Será que troca? */
    int Temp;
    Temp = Num2;
    Num2 = Num1;
    Num1 = Temp;
}
/*------------------------------------------------------*/
int main() {
    int X = 4, Y = 2;

    printf("Antes da Troca, X=%d e Y=%d\n", X, Y);
    Troca(X,Y);
    printf("Depois da Troca, X=%d e Y=%d\n", X, Y);
}
```

Experimente implementar este código. O que se passa quando o tenta executar? Dá o resultado correto? *Não, os dois valores não são trocados…*

Esta função não funciona porque os dois argumentos são passados por valor. Isto significa que a função recebe uma cópia local dos dois parâmetros. Qualquer modificação que é feita nestas cópias locais não se reflete nas variáveis originais.

Em C existe uma técnica que permite resolver este problema. Traduz-se na utilização de ponteiros como parâmetros: em vez de se passar uma cópia do parâmetro, passa-se um ponteiro para esse parâmetro. Este ponteiro pode ser manipulado de forma a permitir alterar o valor da variável por ele apontada (variável externa à função). É curioso salientar que o ponteiro é passado por valor. A função não pode alterar o valor do ponteiro, uma vez que recebe uma cópia local do ponteiro. Contudo, a função pode alterar o conteúdo da memória apontada pelo ponteiro, ou seja, a variável associada a esse ponteiro. A utilização de passagem de parâmetros por ponteiros possui a vantagem de as alterações efetuadas na função se refletirem no próprio valor das variáveis externas à função e no facto de podermos alterar dentro de uma mesma função vários parâmetros simultaneamente. A principal desvantagem na utilização de ponteiros traduz-se na maior complexidade associada à utilização de ponteiros (sobretudo para programadores menos experientes). A seguir é apresentado o exemplo anterior, mas agora utilizando passagem de parâmetros por ponteiros:

```
void Troca(int *Num1, int *Num2) {
    int Temp;

    Temp = *Num2;
    *Num2 = *Num1;
    *Num1 = Temp;
}

/* continua na pag. seguinte */
```

```
int main() {
    int X = 4;
    int Y = 2;

    printf("Antes da Troca, X=%d e Y=%d\n", X, Y);
    Troca(&X,&Y);
    printf("Depois da Troca, X=%d e Y=%d\n", X, Y);
}
```

A passagem de tabelas como argumento de funções é feita por valor (endereço do primeiro elemento da tabela, que é o valor associado ao nome de uma tabela). Assim, a função recebe na realidade um endereço (passagem por valor do endereço ou ponteiro), de forma que, se forem alterados elementos da tabela no corpo da função, essa alteração é permanente, mesmo depois de a função terminar, já que as alterações são efetuadas no próprio valor dos elementos da tabela externa à função:

```
/* Coloca a zero os dim elementos de tab */
void anula( int * tab, int dim) { /* Ou  void anula( int tab[], int dim) */
    for (int i = 0; i < dim; i++)
         tab[i] = 0;
}
/*-----------------------------------------------------------------*/
int main() {
    int tabela[12]={1,2,3,4,5,6,7,8,9,10,11,12};  /* Cria a tabela */
    anula(tabela, 12);   /* Anula todos os elementos da tabela */
    return 0;
}
```

Se a tabela que deve ser passada à função for **multi-dimensional** é necessário **indicar o número de colunas** (pois *os valores são armazenados linha a linha*, e para passar à linha seguinte o compilador precisa de saber o número de colunas).

Considere o seguinte exemplo:

```
#define COL 4   /* num. de colunas da matriz */
/* Anula os elementos da matriz com n linhas e 4 colunas */
void Anula(int tab[][COL], int n) {
   for (int i = 0; i < n; i++)
     for (int j = 0 ; j < COL; j++)
       tab[i][j] = 0;   /* distância ao 1º elemento da tabela: i*4+j */
}
/*-----------------------------------------------------------------*/
int main() {
    /* definição e inicialização */
    int matriz[3][4] = {{0,1,2,3},{4,5,6,7},{8,9,10,11}};
    Anula(matriz, 3);
    return 0;
}
```

1.9 `#include` e múltiplos ficheiros

Em programação há muitas formas de organizar o código de forma a facilitar a vida ao programador:
- comentários (ajuda a ler os programas com maior facilidade);
- documentação (documentar código ajuda a descobrir *bugs*; melhora a clareza);
- indentação (ajuda a perceber o código e a sua estrutura, evitando muitos *bugs*);
- partir funções complexas em várias funções mais simples (diminui a complexidade);
- recorrer a bibliotecas de funções (poupa trabalho ao programador, uso do `#include`);
- distribuir o código por vários ficheiros (ajuda a separar as tarefas, uso de `#include`);
etc.

O programador tem ao seu dispor o uso da diretiva de compilação #include que lhe permite incluir, quer código feito por outros, quer código feito por si próprio.

Em geral, em cada ficheiro de extensão **c** (ou **cpp** para programas em C++) agrupam-se as **definições de funções** relacionadas com uma dada tarefa específica: entrada/saída, manipulação de caracteres, manipulação de tabelas, etc. Para tornar facilmente acessíveis as funções em cada ficheiro **c** às funções noutros ficheiros **c** é também criado um ficheiro de extensão **h**, um ***header file***, que contém os **protótipos** de todas as funções definidas no ficheiro de extensão **c** com o mesmo nome (o nome não precisa de ser o mesmo, mas ajuda o programador).

A diretiva `#include`, como já sabe, permite incluir um ficheiro noutro. Assim, um ficheiro de extensão `principal.c` ao incluir um ficheiro, `tabelas.h` (fazendo: `#include "tabelas.h"`) passa a conter todos os protótipos contidos no ficheiro `tabelas.h`, e seria possível compilar o ficheiro `principal.c`, criando o ficheiro objecto, `principal.obj`. Para criar um programa executável, é necessário *ligar* o código objeto do programa `principal.c` com o código objeto de `tabelas.c`.

A ação de agrupar funções com um tema específico num dado ficheiro de extensão **c** e a criação do respetivo ficheiro com a lista de protótipos permitem tornar os programas modulares e reutilizáveis. Imaginemos que desejávamos agrupar todas a funções associadas com trocas num só ficheiro. Neste momento apenas dispomos de duas funções...

Exemplificando, teríamos o ficheiro `troca.c`:

```
#include "troca.h" /* Assim as funções podem surgir por qualquer ordem */
/*-------------------------------------------------------*/
void Troca(int *Num1, int *Num2) {
    int Temp;
    Temp = *Num2;
    *Num2 = *Num1;
    *Num1 = Temp;
}
/*-------------------------------------------------------*/
void TrocaMal(int Num1, int Num2) {   /* Será que troca? */
    int Temp;
    Temp = Num2;
    Num2 = Num1;
    Num1 = Temp;
}
```

Teríamos também o ficheiro `troca.h`:

```
#ifndef TROCA_H
#define TROCA_H
void Troca(int *Num1, int *Num2);
void TrocaMal(int Num1, int Num2);
#endif
```

As diretivas `#ifndef e #define` são aqui utilizadas para evitar a múltipla inclusão de `troca.h`, que resultaria em erro de compilação, por múltipla declaração dos protótipos das funções aí contidas.

Agora qualquer programa pode usar as funções `troca()`... Por exemplo o programa `principal.c`:

```
#include <stdio.h>
#include "troca.h" /* Equivale a colocar aqui o conteúdo de troca.h */
int main() {
    int X = 4;
    int Y = 2;

    printf("Antes da troca, X=%d e Y=%d\n", X, Y);
    TrocaMal (X,Y);
    printf("Depois da troca (por copia simples), X=%d e Y=%d\n", X, Y);
    Troca(&X,&Y);
    printf("Depois da troca por ponteiro, X=%d e Y=%d\n", X, Y);
}
```

Quando uma equipa escreve software em C, recorre ao `#include` para integrar código feito por várias pessoas. É comum hoje em dia existirem equipas a integrar num único programa mais de 100 contribuições independentes de código! Isto equivale a um programa contendo alguns milhares de funções ou dezenas de milhar de variáveis. Imagine o que seria alguém tentar integrar todo o código num único ficheiro!

O objetivo desta secção é alertá-lo para o facto de num futuro próximo poder ter que recorrer necessariamente à partição de um programa em vários ficheiros de forma a organizar melhor o seu código fonte e permitir a reutilização de algumas funções.